

CaChannel Guide

Geoff Savage
November 1999

A Python class that wraps caPython,
a Python interface to the EPICS channel access protocol.

1	FIRST AND LAST STEPS.....	3
1.1	FIRST STEP.....	3
1.2	LAST STEP.....	3
2	CACHANNEL BY EXAMPLE	3
2.1	SINGLE SYNCHRONOUS ACTIONS	4
2.2	MULTIPLE SYNCHRONOUS ACTIONS.....	4
2.3	ASYNCHRONOUS ACTIONS.....	5
2.4	ASYNCHRONOUS MONITORING.....	7
3	CHANNEL ACCESS BASICS	8
3.1	OVERVIEW.....	8
3.2	FIELD TYPES	8
3.3	ELEMENT COUNTS.....	9
3.4	ERRORS	9
3.5	CALLBACKS.....	10
3.5.1	<i>Connection callback</i>	10
3.5.2	<i>Put callback</i>	10
3.5.3	<i>Get callback</i>	10
3.5.3.1	Original.....	11
3.5.3.2	Status.....	11
3.5.4	<i>Monitor callback</i>	12
3.6	FILE DESCRIPTOR MANAGER.....	12
4	CACHANNEL METHODS	12
4.1	HELPERS	12
4.1.1	<i>getTimeout</i>	12
4.1.2	<i>setTimeout</i>	12
4.1.3	<i>getValue</i>	13
4.2	WAIT	13
4.2.1	<i>searchw</i>	13
4.2.2	<i>putw</i>	13
4.2.3	<i>getw</i>	14
4.3	CONNECTION	14
4.3.1	<i>search_and_connect</i>	14
4.3.2	<i>search</i>	15
4.3.3	<i>clear_channel</i>	15
4.4	WRITE.....	15
4.4.1	<i>array_put</i>	15
4.4.2	<i>array_put_callback</i>	16
4.5	READ.....	16
4.5.1	<i>array_get</i>	16
4.5.2	<i>array_get_callback</i>	16
4.6	EXECUTION.....	17
4.6.1	<i>pend_io</i>	17
4.6.2	<i>pend_event</i>	17
4.6.3	<i>poll</i>	17
4.6.4	<i>flush_io</i>	18
4.7	MONITORING	18
4.7.1	<i>add_masked_array_event</i>	18
4.7.2	<i>clear_event</i>	18
4.8	MACROS	19
4.8.1	<i>field_type</i>	19
4.8.2	<i>element_count</i>	19
4.8.3	<i>name</i>	19
4.8.4	<i>state</i>	20
4.8.5	<i>host_name</i>	20
4.8.6	<i>read_access</i>	20
4.8.7	<i>write_access</i>	20

CaChannel Guide

CaChannel wraps the functions in caPython into a Python class for easier use. CaPython is a Python module that implements the EPICS channel access (CA) communications protocol.

This guide covers:

- CaChannel.py.
- Basics of EPICS channel access.
- Basics of caPython.

Those unfamiliar with EPICS channel access should consult:

- “EPICS R3.12 Channel Access Reference Manual”
- “Channel Access Client Library Tutorial, R3.13”
- The “caPython Guide” explains the functionality of CA available for use in CaChannel.

1 First and Last Steps

1.1 First Step

The first step is accessing the CaChannel module. The python command to do this is:

```
from CaChannel import *
```

This statement imports:

- CaChannel – the CaChannel class.
- CaChannelException – the CaChannel exception object.
- ca – the caPython module.
- dbr_d – a dictionary used to perform CA type conversions. Users do not need to use this dictionary. It is used by CaChannel.

1.2 Last Step

The last step is to close the channels that you are no longer using. This is done automatically when the channel object is deleted. An object created by:

```
obj = CaChannel()
```

is deleted by:

```
del obj
```

2 CaChannel By Example

This chapter reviews the examples that use CaChannel.

- Single synchronous actions – One action (search, get, put) at a time is executed and the results of each action returned to the user before the next action is processed.
- Multiple synchronous actions – Multiple actions are combined into one message. The user must specify when the message is sent for processing and wait for the processing to complete. The results of all the actions are returned to the user.
- Asynchronous actions – Wait for no actions to complete. Instead, flush the actions to be processed and execute a user specified callback routine when the action has completed.

- Asynchronous events – Execute a user specified callback routine when an event occurs in a server.

2.1 Single Synchronous Actions

This example shows the user how to move from caPython v1_5 to v2_0.

Description	v2_0	v1_5
Channel identifier	<code>chid = CaChannel()</code>	<code>chid = ca.connect('pvname')</code>
Connect to a channel	<code>chid.searchw('pvname')</code>	<code>ca.connect('pvname')</code>
Write to a channel	<code>chid.putw(value)</code>	<code>ca.put(chid, value)</code>
Read from a channel	<code>chid.getw(value)</code>	<code>ca.get(chid, value)</code>

Each of the actions (search, put, get) ending with a “w” signify that the action completes before the function returns. In CA terms this means that a call to `ca_pend_io()` is issued to force the action to process and wait for the action to complete. When an exception occurs the offending CA status return is printed using `print ca.message(status)`.

```
#!/bin/env python
#
# filename: ca_wait.py
#
# Test the simple CA methods
#   searchw
#   putw
#   getw
#

from CaChannel import CaChannel
from CaChannel import CaChannelException
import ca

def main():

    # write and read an analog input record
    try:
        catest = CaChannel()
        catest.searchw('catest')
        catest.putw(55)
        print catest.getw()
    except CaChannelException, status:
        print ca.message(status)

    # write and read a waveform record
    try:
        cawave = CaChannel()
        cawave.searchw('cawave')
        l = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
        cawave.putw(l)
        print cawave.getw()
    except CaChannelException, status:
        print ca.message(status)

main()
```

2.2 Multiple Synchronous Actions

Each group of actions (search, put, get) is executed by the call to `pend_io()`. When the actions have completed `pend_io()` returns and the code continues. Each call to

`pend_io()` applies to all the channel objects and not just the object whose method is being invoked. This is a consequence of CA that supersedes OOP. A timeout value (specified in seconds) is associated with a call to `pend_io()` if no timeout is given the default value of one second is used.

Values returned from a `get` are stored for retrieval after `pend_io()` has returned. The returned values are accessed using the `getValue()` method. If `pend_io()` returns an error status then all values read during processing are not guaranteed to be correct.

```
#!/bin/env python
#
# filename: ca_io.py
#
# Test CaChannel io using ca.pend_io()
#   search
#   array_put
#   array_get
#   pend_io
#

from CaChannel import CaChannel
from CaChannel import CaChannelException
import ca

def main():
    try:
        catest = CaChannel()
        cawave = CaChannel()
        scan = CaChannel()

        catest.search('catest')
        cawave.search('cawave')
        scan.search('catest.SCAN')
        catest.pend_io()

        t = (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19)
        catest.array_put(123.456)
        cawave.array_put(t)
        scan.array_put("1 second", ca.DBR_STRING)
        cawave.pend_io()

        catest.array_get()
        cawave.array_get()
        scan.array_get()
        ca.pend_io(1.0)

        print catest.getValue()
        print cawave.getValue()
        print scan.getValue()

        scan.array_get(ca.DBR_STRING)
        scan.pend_io()
        print scan.getValue()

    except CaChannelException, status:
        print ca.message(status)

main()
```

2.3 Asynchronous Actions

Asynchronous execution does not require that the user wait for completion of an action. Instead, a user specified callback function is executed when the action has completed. Each callback takes two arguments:

- `epics_args` – arguments returned from `epics`.
 - `user_args` – arguments specified by the user for use in the callback function.
- Since we don't need to wait for actions to complete we use `flush_io()` instead of `pend_io()` as in the synchronous examples. `Flush_io()` starts execution of actions and returns immediately. To allow the callback mechanism to function the user must call `pend_event()` periodically. `Pend_event()` interrupts the programs main thread to allow completion of outstanding CA activity.

```

#!/bin/env python
#
# filename: ca_cb.py
#
# Test CaChannel callbacks
#     search_and_connect
#     array_put_callback
#     array_get_callback
#     flush_io
#     pend_event
#
from CaChannel import CaChannel
from CaChannel import CaChannelException
import ca
import time
import sys

def connectCb(epics_args, user_args):
    print "connectCb: Python connect callback function"
    print type(epics_args)
    print epics_args
    print user_args

def putCb(epics_args, user_args):
    print "putCb: Python put callback function"
    print type(epics_args)
    print epics_args
    print ca.name(epics_args['chid'])
    print ca.dbr_text(epics_args['type'])
    print epics_args['count']
    print ca.message(epics_args['status'])
    print user_args

def getCb1(epics_args, user_args):
    print "getCb: Python get callback function"
    print type(epics_args)
    print epics_args
    print "pvName = ", ca.name(epics_args['chid'])
    print "type = ", ca.dbr_text(epics_args['type'])
    print "count = ", epics_args['count']
    print "status = ", ca.message(epics_args['status'])
    print "user args = ", user_args
    print "value(s) = ", epics_args['pv_value']

def getCb2(epics_args, user_args):
    print "getCb: Python get callback function"
    print type(epics_args)
    print epics_args
    print "pvName = ", ca.name(epics_args['chid'])
    print "type = ", ca.dbr_text(epics_args['type'])
    print "count = ", epics_args['count']
    print "status = ", ca.message(epics_args['status'])
    print "user args = ", user_args
    print "value(s) = ", epics_args['pv_value']
    print ca.alarmSeverityString(epics_args['pv_severity'])
    print ca.alarmStatusString(epics_args['pv_status'])

def main():

```

```

try:
    chan = CaChannel()
    print "search_and_connect"
    chan.search_and_connect('catest', connectCb)
    chan.flush_io()
    for i in range(20):
        chan.pend_event()
    print "put_callback"
    chan.array_put_callback(3.3, None, None, putCb)
    chan.flush_io()
    for i in range(20):
        chan.pend_event()
    print "get_callback"
    chan.array_get_callback(None, None, getCb1)
    chan.flush_io()
    for i in range(20):
        chan.pend_event()
    print "get_callback with status"
    chan.array_get_callback(ca.dbf_type_to_DBR_STS(chan.field_type()),
None, getCb2)
    chan.flush_io()
    for i in range(20):
        chan.pend_event()
except CaChannelException, status:
    print ca.message(status)

try:
    cawave = CaChannel()
    print "cawave: search_and_connect"
    cawave.search_and_connect('cawave', connectCb)
    cawave.flush_io()
    for i in range(20):
        cawave.pend_event()
    print "cawave: array_put_callback"
    l = [0,1,2,3,4,5,6,7,8,9]
    cawave.array_put_callback(l, None, None, putCb)
    cawave.flush_io()
    for i in range(20):
        cawave.pend_event()
    print "cawave: array_get_callback"
    cawave.array_get_callback(ca.dbf_type_to_DBR_STS(cawave.field_type
()), None, getCb2)
    cawave.flush_io()
    for i in range(20):
        cawave.pend_event()
except CaChannelException, status:
    print ca.message(status)

main()

```

2.4 Asynchronous Monitoring

Watch for changes in value or alarm state of a process variable. A callback is executed when a change is seen.

```

#! /bin/env python
#
# filename: ca_event.py
#
# Test CaChannel monitors
#   search
#   pend_io
#   fdmgr_start()
#   fdmgr_pend
#   add_masked_array_event
#

```

```

from CaChannel import CaChannel
from CaChannel import CaChannelException
import ca
import time
import sys

def eventCb(epics_args, user_args):
    print "eventCb: Python callback function"
    print type(epics_args)
    print epics_args
    print ca.message(epics_args['status'])
    print "new value = ", epics_args['pv_value']
    print ca.alarmSeverityString(epics_args['pv_severity'])
    print ca.alarmStatusString(epics_args['pv_status'])

def main():
    ca.fdmgr_start()
    try:
        chan = CaChannel()
        print 'search for "catest"'
        chan.search('catest')
        chan.pend_io()
    except CaChannelException, status:
        print ca.message(status)

    try:
        print 'add event'
        chan.add_masked_array_event(ca.dbf_type_to_DBR_STS(chan.field_type
()), None, ca.DBE_VALUE | ca.DBE_ALARM, eventCb)
    except CaChannelException, status:
        print ca.message(status)

    while 1:
        ca.fdmgr_pend()

main()

```

3 Channel Access Basics

EPICS channel access (CA) is the communication protocol used to transfer information between EPICS servers and clients.

3.1 Overview

Channel access provides access to process variables (PV) that are accessible from EPICS channel access servers. Process variables are fields in EPICS records. (This definition is expanded when using the portable channel access server.). The steps for interacting with a process variable are:

1. Connect – create a connection between your application and a process variable.
2. Read (as needed)– read data held in the process variable.
3. Write (as needed)– write data to the process variable.
4. Close – close the connection between an application and a process variable.

3.2 Field types

Each field in an EPICS record has a native type. The native types are listed in the following table along with the C type to which they correspond.

Native Type	Request Type	C Type	Python Type
-------------	--------------	--------	-------------

ca.DBF_STRING	ca.DBR_STRING	array of char	String
ca.DBF_CHAR	ca.DBR_CHAR	char	String
ca.DBF_ENUM	ca.DBR_ENUM	int	Int
ca.DBF_SHORT	ca.DBR_SHORT	short (16 bits)	Int
ca.DBF_INT	ca.DBR_INT	short (16 bits)	Int
ca.DBF_LONG	ca.DBR_LONG	long (32 bits)	Int
ca.DBF_FLOAT	ca.DBR_FLOAT	float	Float
ca.DBF_DOUBLE	ca.DBR_DOUBLE	double	Float

This table also lists the EPICS request types. Users can request that the type of the read or written value be changed internally by EPICS. Typically this adds a time penalty and is not recommended.

The one area where type conversion is extremely useful is dealing with fields of type ca.DBF_ENUM. An ENUM value can only be one from a predefined list. A list consists of a set of string values that correspond to the ENUM values (similar to the C enum type). It is easier to remember the list in terms of the strings instead of the numbers corresponding to each string.

The native type of a channel is obtained using the `field_type()` method.

```
chan = caChannel()
chan.searchw('pvname')
fieldType =chan.field_type()
```

Convert the field type to a string using either the `dbf_text()` or `dbr_text()` methods.

```
print ca.dbf_text(fieldType)
print ca.dbr_text(fieldType)
```

3.3 Element counts

Each data field can contain one or more data elements. The number of data elements is referred to as the native element count for a field. The number of data elements written to or read from a data field with multiple elements is user controllable. All or some data elements can be read. When some data elements are accessed the access is always started at the first element. It is not possible to read part of the data and then read the rest of the data.

The native element count for a channel is determined using the `element_count()` method.

```
chan = caChannel()
chan.searchw('pvname')
elementCount =chan.element_count()
```

3.4 Errors

All CA errors are converted to exceptions that also provide access to the CA status.

```
try:
    (Execute CA calls here)
except CaChannelException, status:
    print ca.message(status)
```

The different status values are accessed through `ca.ECA_XXXX` values and correspond to the value returned from the underlying CA C function.

3.5 Callbacks

Callbacks come in four types:

- Connection
- Put
- Get
- Monitor

In each case, a callback function must have two arguments. If two arguments are not specified then the callback will not function.

- `epics_args` – arguments returned from epics.
- `user_args` – arguments specified by the user for use in the callback function.

`User_args` are returned as a tuple. It is the users responsibility to understand how to access `user_args`. Accessing value in `epics_args` is explained below.

3.5.1 Connection callback

`Epics_args` is a two-element tuple. The first item in the tuple is the channel identifier for the channel just connected and the second item is the channels connection state. The possible channel channels connection states are `ca.CA_OP_CONN_UP` and `ca.CA_OP_CONN_DOWN`. An example callback function:

```
def connectCb(epics_args, user_args):
    print "connectCb: Python connect callback function"
    print type(epics_args)
    print epics_args
    print user_args
```

Usage:

```
chan.search_and_connect('pvname', connectCb)
```

3.5.2 Put callback

`Epics_args` is a dictionary with the following keys.

- `epics_args['chid']` = channel identifier.
- `epics_args['type']` = requested type for the operation. (One of `ca.DBR_XXXX`)
- `epics_args['count']` = request count for the operation.
- `epics_args['status']` = CA status code from the server performing the operation.

```
def putCb(epics_args, user_args):
    print "putCb: Python put callback function"
    print type(epics_args)
    print epics_args
    print ca.name(epics_args['chid'])
    print ca.dbr_text(epics_args['type'])
    print epics_args['count']
    print ca.message(epics_args['status'])
    print user_args
```

3.5.3 Get callback

Get callbacks come in a variety of flavors:

- Original (implemented) – return the values requested.

- Status (implemented) – return the values requested and the alarm status and severity.
- Time (not implemented) – add the time stamp of the PV to the values returned by Status.
- Graphics (not implemented) – return the values requested, alarm status and severity, engineering units, upper and lower display limits, and the alarm limits.
- Control (not implemented) – add the control limits to the values returned by Graphics.

3.5.3.1 Original

These values are returned when one of `ca.DBR_XXXX` is specified for the database request type. `Epics_args` is a dictionary with the following keys..

- `epics_args['chid']` = channel identifier.
- `epics_args['type']` = requested type for the operation. (One of `ca.DBR_XXXX`)
- `epics_args['count']` = request count for the operation.
- `epics_args['status']` = CA status code from the server performing the operation.
- `epics_args['pv_value']` = data returned by the server. Multiple data elements are returned in a tuple.

```
def getCb1(epics_args, user_args):
    print "getCb: Python get callback function"
    print type(epics_args)
    print epics_args
    print "pvName = ", ca.name(epics_args['chid'])
    print "type = ", ca.dbr_text(epics_args['type'])
    print "count = ", epics_args['count']
    print "status = ", ca.message(epics_args['status'])
    print "user args = ", user_args
    print "value(s) = ", epics_args['pv_value']
```

3.5.3.2 Status

These values are returned when one of `ca.dbf_type_to_DBR_STS` (`ca.DBR_XXXX`) is specified for the database request type. `Epics_args` is a dictionary with the following keys..

- `epics_args['chid']` = channel identifier.
- `epics_args['type']` = requested type for the operation. (One of `ca.DBR_XXXX`)
- `epics_args['count']` = request count for the operation.
- `epics_args['status']` = CA status code from the server performing the operation.
- `epics_args['pv_value']` = data returned by the server. Multiple data elements are returned in a tuple.
- `epics_args['pv_severity']` = the pv's current alarm severity.
- `epics_args['pv_status']` = the pv's current alarm status.

```
def getCb2(epics_args, user_args):
    print "getCb: Python get callback function"
    print type(epics_args)
    print epics_args
    print "pvName = ", ca.name(epics_args['chid'])
    print "type = ", ca.dbr_text(epics_args['type'])
    print "count = ", epics_args['count']
    print "status = ", ca.message(epics_args['status'])
    print "user args = ", user_args
    print "value(s) = ", epics_args['pv_value']
    print ca.alarmSeverityString(epics_args['pv_severity'])
    print ca.alarmStatusString(epics_args['pv_status'])
```

3.5.4 Monitor callback

EPICS can monitor PVs for three types of events:

1. When the value changes by more than the monitor dead band.
2. When the value changes by more than the archiver dead band.
3. When the alarm status of the PV changes.

The values returned in a callback match those of the other callbacks depending on the database request type. The monitoring functions are specified by or'ing the following values.

Monitor Mask	Description
ca.DBE_VALUE	when the channel's value changes by more than MDEL
ca.DBE_LOG	when the channel's value changes by more than ADEL
ca.DBE_ALARM	when the channel's alarm state changes

3.6 File Descriptor Manager

4 CaChannel Methods

4.1 Helpers

Methods added to the interface to make it easier to use.

4.1.1 getTimeout

Synopsis

```
timeout = chan.getTimeout()
```

Arguments

None

Comments

If a timeout is not specified as an argument when using methods requiring a timeout value the default timeout value is used.

Returns

timeout (Float): value of the current default timeout.

4.1.2 setTimeout

Synopsis

```
chan.setTimeout(timeout)
```

Arguments

timeout (float): new default timeout value

Comments

If a timeout is not specified as an argument when using methods requiring a timeout value to be specified the default timeout value is used.

Exceptions

ValueError – timeout must be greater than or equal to 0 (wait forever)

4.1.3 **getValue**

Synopsis

```
val = chan.getValue()
```

Arguments

None

Comments

Used to retrieve values read from a PV after a read has completed successfully. See `array_get()`.

4.2 **Wait**

Each CaChannel wait method makes a channel access function call followed by a call to `ca.pend_io()`. The effect is that each action, search, put, and get, is instructed to execute. The call to `pend_io()` halts program execution and waits for the action to complete or the wait times out.

These methods are designed to be used in simple applications and provide novice users with a simple set of methods to use while becoming acquainted with channel access.

4.2.1 **searchw**

Synopsis

```
chan.searchw('pvName')
```

Arguments

`pvName` (string): process variable to connect to

Description

Attempt to establish and maintain a virtual circuit between the caller's application and a process variable.

Comments

Find and open a connection to a process variable

4.2.2 **putw**

Synopsis

```
chan.putw(value, [req_type])
```

Arguments

Value – value(s) to be sent
`req_type` – database request type for write

Description

Write a value or array of values to a channel.

Comments

Write a value or values to a process variable. If the request type is omitted the data is written as the Python type corresponding to the native format. Multi-element data is specified as a tuple or a list. Internally the sequence is converted to a list before inserting the values into a C array. Access using non-numerical types is restricted to the first element in the data field. Mixing character types with numerical types writes bogus results

but is not prohibited at this time. DBF_ENUM fields can be written using DBR_ENUM and DBR_STRING types. DBR_STRING writes of a field of type DBF_ENUM must be accompanied by a valid string of the possible enumerated values.

4.2.3 getw

Synopsis

Value = chan.getw([req_type])

Arguments

Req_type (optional) – database request type for read

Comments

Read a value or values from a process variable. If the request type is omitted the data is returned to the user as the Python type corresponding to the native format. Multi-element data has all the elements returned as items in a list and must be accessed using a numerical type. Access using non-numerical types is restricted to the first element in the data field. Mixing character types with numerical types returns bogus results but is not prohibited at this time. DBF_ENUM fields can be read using DBR_ENUM and DBR_STRING types. DBR_STRING reads of a field of type DBF_ENUM returns the string corresponding to the current enumerated value.

Returns

Value – value(s) read

Exceptions

CaChannelException, status

4.3 Connection

Methods that create and delete connections with process variables.

4.3.1 search_and_connect

Synopsis

chan.search_and_connect('pvName', callback, *user_args)

Arguments

pvName (string): process variable to connect to
callback (function): function called when the connection is made
*user_args: variable number of user arguments that are passed to callback when it is invoked

Description

Attempt to establish and maintain a virtual circuit between the caller's application and a process variable.

Comments

None

4.3.2 search

Synopsis

```
chan.search('pvName')
```

Arguments

pvName (string): process variable to connect to

Description

Attempt to establish and maintain a virtual circuit between the caller's application and a process variable.

Comments

None

4.3.3 clear_channel

Synopsis

```
chan.clear_channel()
```

Arguments

None

Description

Close a channel created by `search()` or `search_and_connect()`.

Comments

Clearing a channel does not cause its disconnect handler to be called. Clearing a channel does remove any events (monitors) registered for that channel. If the channel is currently connected then resources are freed only some time after this request is flushed out to the server.

4.4 Write

Write data to process variables.

4.4.1 array_put

Synopsis

```
chan.array_put(value, [req_type], [count])
```

Arguments

value: data to be written. For multiple value use a list or tuple.
req_type (int, optional): database request type, must be compatible with the data type of value. Defaults to the native data type for the channel.
count (int, optional): number of data values to write. Defaults to the native count for the channel.

Description

Write a value or array of values to a channel.

Comments

None

4.4.2 array_put_callback

Synopsis

```
chan.array_put_callback(value, req_type, count, callback, *user_args)
```

Arguments

value: data to be written. For multiple values, use a list or tuple.

req_type (int, optional): database request type, must be compatible with the data type of value.

count (int, optional): number of data values to write.

callback (function): function called when the write is completed

*user_args: variable number of user arguments that are passed to callback when it is invoked

Description

Write a value or array of values to a channel and execute the user-supplied callback after the put has completed.

Comments

If the value of None is specified for req_type or count then the default value for the channel is used.

4.5 Read

Read data from process variables.

4.5.1 array_get

Synopsis

```
chan.array_get([req_type], [count])
```

Arguments

req_type (int, optional): database request type, must be compatible with the data type of value. Defaults to the native data type for the channel.

count (int, optional): number of data values to write. Defaults to the native count for the channel.

Description

Read a value or array of values from a channel.

Comments

Once the get has been completed (after the successful completion of a `pend_io()`), the data is retrieved by a call to the `getValue` method.

```
val = chan.getValue()
```

4.5.2 array_get_callback

Synopsis

```
chan.array_get_callback(req_type, count, callback, *user_args)
```

Arguments

req_type (int, optional): database request type, must be compatible with the data type of value.

count (int, optional): number of data values to write.

callback (function): function called when the write is completed

*user_args: variable number of user arguments that are passed to callback when it is invoked

Description

Read a value or array of values from a channel and execute the user-supplied callback after the get has completed.

Comments

If the value of None is specified for req_type or count then the default value for the channel is used.

4.6 Execution

CA actions are buffered until the buffer is full or one of these methods is executed at which time the actions in the buffer are sent in a message for execution.

4.6.1 pend_io

Synopsis

```
chan.pend_io([timeout])
```

Arguments

timeout (float, optional, seconds) – length of time to wait for calls to complete

Description

Flush the send buffer and wait until outstanding queries complete or the specified timeout expires.

Comments

If a timeout is not specified the default timeout is used. The default timeout is manipulated with setTimeout() and getTimeout().

4.6.2 pend_event

Synopsis

```
chan.pend_event([timeout])
```

Arguments

timeout (float, optional) – length of time to wait for calls to complete

Description

Flush the send buffer and wait for asynchronous events for timeout seconds.

Comments

If a timeout is not specified a timeout of 0.1 seconds is used. A timeout of zero waits forever.

4.6.3 poll

Synopsis

```
chan.poll()
```

Arguments

None

Description

Calls `pend_event()` with a timeout short enough to poll.

Comments

Outstanding channel access background activity executes during the poll.

4.6.4 `flush_io`

Synopsis

```
chan.flush_io()
```

Arguments

None

Description

Flush the send buffer.

Comments

None

4.7 *Monitoring*

4.7.1 `add_masked_array_event`

Synopsis

```
chan.add_masked_array_event(req_type, count, mask, callback,  
*user_args)
```

Arguments

`req_type` (int, optional): database request type, must be compatible with the data type of value.

`count` (int, optional): number of data values to write.

`mask` (int): logical or of `ca.DBE_VALUE`, `ca.DBE_LOG`, or `ca.DBE_ALARM`

`callback` (function): function called when the write is completed

`*user_args`: variable number of user arguments that are passed to callback when it is invoked

Description

Specify a callback function to be executed whenever significant changes occur on a channel.

Comments

Any callback currently registered is replaced by the new callback. Only one callback can be registered for monitoring on a channel. If the value of None is specified for `req_type` or `count` then the default value for the channel is used. See the discussion of the monitor callback for more information on mask.

4.7.2 `clear_event`

Synopsis

```
chan.clear_event()
```

Arguments

None

Description

Remove a callback function that is executed whenever significant changes occur on a channel.

Comments

Only one callback can be registered for monitoring on a channel.

4.8 Macros

4.8.1 field_type

Synopsis

```
FieldType = chan.field_type()
```

Arguments

None

Comments

None

Returns

fieldType (int) – native field type

4.8.2 element_count

Synopsis

```
ec = chan.element_count()
```

Arguments

None

Comments

None

Returns

ec (int) – native element count

4.8.3 name

Synopsis

```
pvName = chan.name()
```

Arguments

None

Comments

None

Returns

pvName (string) – channel name specified when the channel was connected

4.8.4 state

Synopsis

state = chan.state()

Arguments

None

Comments

cs_ - 'channel state'

ca.cs_never_conn valid chid, IOC not found

ca.cs_prev_conn valid chid, IOC was found, but unavailable

ca.cs_conn valid chid, IOC was found, still available

ca.cs_closed invalid chid

Returns

state (int) – current state of the connection

4.8.5 host_name

Synopsis

hostName = chan.host_name()

Arguments

None

Comments

None

Returns

hostName (string) – host name that houses the process variable

4.8.6 read_access

Synopsis

access = chan.read_access()

Arguments

None

Comments

None

Returns

access (int) – TRUE if channel can be read, FALSE otherwise

4.8.7 write_access

Synopsis

access = chan.write_access()

Arguments

None

Comments

None

Returns

access (int) – TRUE if channel can be written, FALSE otherwise